**Linux Namespace**

In Linux, a **namespace** is a mechanism for isolating system resources so that different processes or containers can operate in separate environments without interfering with each other. Each namespace type provides a different aspect of isolation for system resources.

**cgroups** and **namespaces** are powerful tools for managing resources and isolating processes in Linux systems

**Cgroup Namespace (cgroup)**: Isolates the view of control groups (cgroups), which helps in managing resources like CPU, memory, and I/O.

**Process ID Namespace (pid)**: Provides a separate process ID number space, so processes can have the same PID in different namespaces without conflict.

**Container**

**Containers** are a lightweight, portable, and efficient way to package and run applications and their dependencies in isolated environments, Unlike virtual machines (VMs), which virtualize the entire operating system, containers virtualize only the application and its dependencies, sharing the host OS kernel while maintaining isolation.

**Key Characteristics of Containers**

**Isolation**: Containers provide process and file system isolation, meaning that processes running inside a container cannot directly interact with processes running in other containers or on the host system. This is achieved using Linux features like namespaces and cgroups.

**Portability**: Containers bundle an application with its dependencies, configuration files, and libraries. This ensures that the application runs consistently across different environments, from development to production.

**Lightweight**: Containers share the host OS kernel and only package the necessary binaries and libraries. This makes them more efficient in terms of resource usage compared to virtual machines, which require a full OS instance.

**Fast Startup**: Containers start almost instantly because they do not require booting a full OS. This is particularly advantageous for scaling applications up and down quickly.

**Scalability**: Containers can be easily replicated and managed across a cluster of machines. Container orchestration tools like Kubernetes help manage scaling, load balancing, and deployment.

**Version Control**: Container images can be versioned and stored in container registries. This allows developers to roll back to previous versions or deploy specific versions of their applications.

**Components of a Container**

**Container Image:**

Definition: A container image is a static, read-only snapshot of a filesystem that includes the application, its dependencies, and the environment configuration.

Format: Container images are usually built using a Dockerfile or similar configuration files.

Storage: Images are stored in container registries (like Docker Hub, Google Container Registry) and can be pulled and deployed as needed.

**Container Runtime:**

Definition: The container runtime is the software responsible for running containers. It handles tasks such as starting, stopping, and managing containers.

Examples: Docker Engine, containerd, and Podman are popular container runtimes.

**Container Engine:**

Definition: A container engine is a higher-level tool that provides a user-friendly interface for managing containers. It includes the container runtime as well as additional features like image building and management.

Examples: Docker is a widely used container engine that provides a complete suite of tools for container management.

**Container Orchestration:**

Definition: Container orchestration involves managing the deployment, scaling, and operation of containerized applications across a cluster of machines.

Examples: Kubernetes, Docker Swarm, and Apache Mesos are popular container orchestration platforms.

**Docker Commands**

**Image Commands**

docker image – Listing the images

docker rmi <image name> - Removing image

docker pull <image name> - Pulling the image from the repository

docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG] - Tag an Image

**Container Commands**

docker ps

docker ps -a

docker run [OPTIONS] IMAGE [COMMAND] [ARG...] - Run a Container

docker stop < CONTAINER ID/ CONTAINER Name> - Stop a Container

docker start < CONTAINER ID/ CONTAINER Name> - Start a Container

docker exec -it < CONTAINER ID/ CONTAINER Name> bash - login to the container with bash shell

docker logs < CONTAINER ID/ CONTAINER Name> - checking the logs

docker inspect < CONTAINER ID/ CONTAINER Name> - Inspect a Container

**Network Commands**

docker network ls – Listing networks

docker network create [OPTIONS] NETWORK Name - Create a network

docker network inspect <NETWORK NAME> - Inspect a Network

docker network rm NETWORK [NETWORK...] - Remove a Network

**Build and Run docker image:**

docker build -t <image\_name:latest> . - Build a docker image from current directory Dockerfile

-t is for tag and image with name

docker run -it -d -p 8000:8000 <image name>

options

-it- Interactive, -d – detach mode and -p Map container ports to host ports

Eg: With Variables and variable files.

docker run -d --name unni\_post\_db\_contaner --network db\_net -e POSTGRES\_USER=postgres -e POSTGRES\_PASSWORD=mysecretpassword -e POSTGRES\_DB=postgres postgres

docker run -d --name unni\_post\_app\_container --network db\_net --network app\_net -e POSTGRES\_USER=postgres -e POSTGRES\_PASSWORD=mysecretpassword -e POSTGRES\_DB=postgres -e POSTGRES\_HOST=172.20.0.2 -p 8000:5000 testapp2

docker run -d --name my\_app\_container --network app\_network --network db\_network --env-file .env my\_app\_image

**GIT**

Git - Version Control System (VCS) - Track the changes made on codes

git init - Initialize a blank repo, a new Git repo and .git directory will be created, and a Master Branch.

git commit - Save changes to the local repo, This command helps to track the record of all the changes made.

git add - This is to add a specific file from directory to the staging area

git status - This is to display the status of current repo and staging area, This will show what is happening with git add and git commit and It shows the untracked files

git merge - Used to integrate different branches into one branch.

git push - Push local repository changes to a remote repo

git pull - Fetch and merge changes from remote repo to local branch. ( git fetch+git merge)

git clone - Create a clone or copy of a target/remote repo

git branch - Like a tree, this will list all local branches, Independent line of developement

git checkout - Works together with git branch, Uses to navigate between branches.

git config - command for configuration options for Git

git diff - compare the different states of a files in repo

git log - shows the history of commits, It shows everything happening to a repo.

git reset - undo the change state of a local git repo

git rebase - rewrite commit history and integrate changes from one branch to another.

**Configured the git access to the Git repo**

git config --global user.email [krishna.tva@gmail.com](mailto:krishna.tva@gmail.com)

git config --global user.name "krishnatva"

git config –list

configured the remote repo and ssh connection.

# git remote -v

origin git@github.com:krishnatva/Todo\_app.git (fetch)

origin git@github.com:krishnatva/Todo\_app.git (push)

git remote set-url origin git@github.com:krishnatva/Todo\_app.git

echo "# Todo\_app" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin git@github.com:krishnatva/Todo\_app.git

git push -u origin main

**Created a test file and edited the contents**

# git add test

# git status

On branch main

Your branch is ahead of 'origin/main' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: test

git commit -m "unni second commit"

git push -u origin main

created branch and checkout to branch, and modified the files from terminal and GUI, done pull request.

git branch unni\_app\_branch

# git branch

\* main

unni\_app\_branch